Interfacing OpenAD and Tapenade

Sri Hari Krishna Narayanan*, Laurent Hascoët†

1 Introduction

Development of a capable algorithmic differentiation (AD) tool requires large developer effort to provide the various flavors of derivatives, to experiment with the many AD model variants, and to apply them to the candidate application languages. Considering the relatively small size of the academic teams that develop AD tools, collaboration between them is a natural idea. This collaboration can exist at the level of research ideas as well as tool development.

This work describes the interoperation of two source to source transformation AD tools OpenAD [1, 2] and Tapenade [3, 4]. The interoperative pipeline uses the parsing and source analysis capabilities of Tapenade with the transformation algorithms of OpenAD.

The aim of such interoperability is to ensure the robustness and stability of the AD tools. The redundancy between some components of either tool would offer more flexibility to the end-user. A weakness in one component may be compensated by choosing another route in the components graph. In the same order of ideas, a long-term objective is "à la carte" AD, where one may combine powerful capabilities from either tools for instance, the preaccumulation capacities of OpenAD with the accurate data-flow model of Tapenade for activity, adjoint liveness, and TBR analysis. Additionally, not relying on any one component such as the front-end compiler developed externally, allows the AD tool to persist beyond the lifetime of that front end compiler. Even further, we can analyze the strengths and weaknesses of each tool's AD model. These models are quite close (source transformation, with a store-on-kill adjoint model), yet some choices differ, for instance association-by-name vs. association-by-address [5].

2 Architecture of the interoperable tool

Interoperation between the OpenAD and Tapenade is possible because they share the same global architecture i.e. a front-end which parses and builds an internal representation, followed by a static data-flow analysis stage, then actual building of the differentiated program still in internal form, and finally a back-end that outputs this differentiated internal form into new source files. Figure 1 details this architecture for OpenAD and for Tapenade.

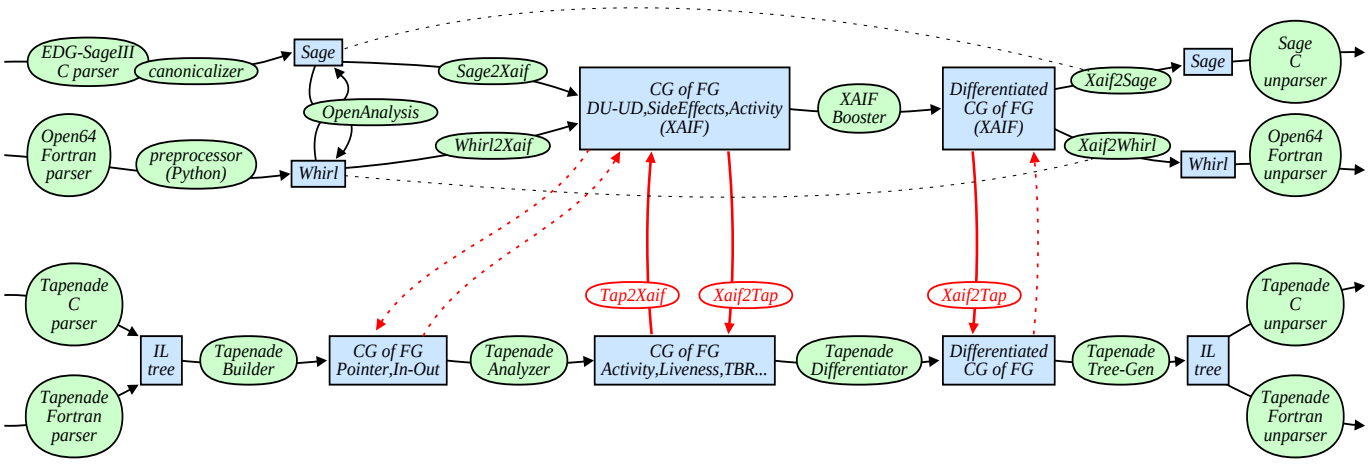

Figure 1: Compared architectures and bridges between OpenAD (top) and Tapenade (bottom)

OpenAD pipeline OpenAD's pipeline starts with a custom Python preprocessor. The preprocessed code is parsed using an Open64 translator called `whirl2xaif` that generates whirl intermediate representation, invokes OpenAnalysis for program analysis, and transforms the whirl representation and analysis results into the XML Abstract Interface Form (XAIF) [6]. The XAIF is given to XAIFBooster [7] as input which produces an augmented (differentiated) XAIF. A stored version of the original whirl representation and the differentiated XAIF are used by `xaif2whirl` to

*Corresponding Author, Argonne National Laboratory, snarayan@mcs.anl.gov
†INRIA Sophia-Antipolis, laurent.hascoet@inria.fr

produce a differentiated whirl representation which is then used to generate Fortran code. Then, Fortran code is post processed using a python post processor. The pipeline for C programs is similar, using a different parser/unparser.

Tapenade pipeline Tapenade pipeline starts with a parser for the application language. Parsers produce an abstract syntax tree with predefined tree operators encompassing the syntactic structures of both Fortran and C, extensible to other Imperative Languages, hence the name "IL". The input IL tree is immediately transformed into Tapenade's internal representation, a Call Graph (CG of FG) whose nodes represent procedures, and each procedure node is itself a Flow Graph whose nodes are Basic Blocks of sequential code. The data-flow analysis stage operates on the CG of FG, annotating it with the analysis results, then actual differentiation produces a new CG of FG, which is then translated back into an IL tree. The last stage unparses the differentiated IL tree into the application language.

This architecture adopted by both OpenAD and Tapenade is standard and is used by many other AD tools. Having the same architecture, however, does not ensure interoperability. At a deeper level, it is also necessary that the internal representation of programs use the same concepts so that it can be easily transferred between tools. OpenAD and Tapenade explicitly use a CG of FG structure. Symbol Tables are also used in the same manner.

Another requirement is that the tools must not be monolithic: successive components of their work flow must be identified and clearly separated. For OpenAD, each component represented by the blue rectangular boxes in fig. 1 is programmed independently, and implemented in an arbitrary language. Each component respects the imposed format of its input/output. At two places, this format is XAIF. The XAIF holds the call graph, flow graph, and symbol tables, the XAIF also contains the results of alias and analysis as well as DefUse chains and DefOverwrite chains. Some components, however, require information from tools earlier in the work-flow. These (dotted lines in fig. 1) are propagated through the chain in an ad-hoc manner, as pointers kept in XAIF. This restricts modularity, as some components downstream impose a specific component upstream. Tapenade was not designed with openness as a primary objective. The work-flow components, however, are clearly separated Java packages, but they operate on an in-memory internal representation of the code. At most places, this in-memory object is the Call Graph of Flow Graphs discussed above, except at both ends of the chain, where it is an abstract syntax tree in the IL formalism.

```
<xaif:CallGraph
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xaif="http://www.mcs.anl.gov/XAIF"
   xsi:schemaLocation="http://www.mcs.anl.gov/XAIF xaif.xsd"
   program_name="Test" ...>
 <xaif:ScopeHierarchy>
  <xaif:Scope vertex_id="Root" ...>
        ...
  </xaif:Scope>
  <xaif:Scope vertex_id="Scope5" ...>
   <xaif:SymbolTable>
    <xaif:Symbol symbol_id="x" kind="variable" active="1" type="real" shape="scalar"/>
        ...
   </xaif:SymbolTable>
  </xaif:Scope>
        ...
  <xaif:ScopeEdge ... source="Scope2" target="Root" />
  <xaif:ScopeEdge ... source="Scope5" target="Scope2" />
 </xaif:ScopeHierarchy>
 <xaif:AliasSetMap> ... </xaif:AliasSetMap>
 <xaif:DUUDSetMap>  ... </xaif:DUUDSetMap>
 <xaif:DOSetMap>    ... </xaif:DOSetMap>
 <xaif:ControlFlowGraph vertex_id="Unit1" symbol_id="head" scope_id="Scope1" ...>
  <xaif:ArgumentList>
        ...
  </xaif:ArgumentList>
  <xaif:Entry vertex_id="entry" .../>
  <xaif:BasicBlock vertex_id="B0" scope_id="Scope5" ...>
   <xaif:Assignment ...>
    <xaif:AssignmentLHS> ... </xaif:AssignmentLHS>
    <xaif:AssignmentRHS> ... </xaif:AssignmentRHS>
   </xaif:Assignment>
   <xaif:SubroutineCall symbol_id="foo" formalArgCount="2" ...>
    <xaif:Argument position="1"> ... </xaif:Argument>
    <xaif:Argument position="2"> ... </xaif:Argument>
   </xaif:SubroutineCall>
  </xaif:BasicBlock>
  <xaif:Exit vertex_id="exit" .../>
  <xaif:ControlFlowEdge ... source="entry" target="B0"/>
  <xaif:ControlFlowEdge ... source="B0" target="exit"/>
 </xaif:ControlFlowGraph>
 <xaif:ControlFlowGraph vertex_id="Unit2" symbol_id="foo" scope_id="Scope1" ...>
        ...
 </xaif:ControlFlowGraph>
 <xaif:CallGraphEdge ... source="Unit1" target="Unit2"/>
</xaif:CallGraph>
```

Figure 2: Overview of XAIF Callgraph Schema

Based on the architecture of the two tools the connections where program information between them can be exchanged can be identified (red arrows in fig. 1). The connections represented by solid arrows are the ones implemented and tested in this work. Starting from either tool's front-end, one may transfer to the other tool to take advantage of its additional analyses, then run one tool's differentiation component and finish with either tool's back-end. At present, because of the dependencies of OpenAD's back-ends on their respective front-ends, selecting an OpenAD back-end (rightmost bridge arrow in fig. 1) implies that the corresponding OpenAD front-end (leftmost bridge arrow) must have been used. The format used for transfers between separate processes must be XAIF, and it is Tapenade's task to transform its in-memory internal representation to XAIF and back. Figure 1 is simplified and does not show possible connections upstream of the OpenAnalysis tool(s). In the future we may be able to split up components of the actual differentiation stages, allowing for fine blending of the differentiation models.

```
Tap2Xaif.sh head.f -o head.tappre.xaif
XAIFBOOSTER_BASE/algorithms/BasicBlockPreaccumulationReverse/driver/oadDriver -p -v -i head.tappre.xaif -o head.tappre.xbr.xaif -s {
     OPENADROOT}/xaif/schema/ -c {OPENADROOT}/xaif/schema/examples/inlinable_intrinsics.xaif
Xaif2Tap.sh head.tappre.xbr.xaif -o head.tappre.xbr.tappost.f
```

Figure 3: Steps involved in generating output code

3 Translating to and from XAIF

Fig. 2 shows an overview of the CallGraph element of the XAIF schema that represents a program. Within it, analysis results for DefOverwrite chains, DefUse chains, Alias analysis results, and Scope hierarchy exist along with the control flow graphs for each procedure in the program. Activity analysis results are stored within the symbol tables inside the scope hierarchy, and within the statements contained inside the control flow graphs.

Translating between XAIF and Tapenade's internal representation is straightforward. There is a natural match between almost all structures, which is not so surprising. Obviously, graph structures require some easy serialization to translate to XAIF. One difference between the tools is that XAIF systematically uses a general graph structure even for abstract syntax trees, which makes it a little harder to find the root of these trees. However, this allows XAIF to naturally represent common sub-expressions. Translation to Tapenade loses this and results in duplicated common sub-expressions.

Tree operators for the abstract syntax of general imperative languages are almost the same in XAIF and Tapenade. However, while Tapenade tries to handle the union of operators used in Fortran and in C, OpenAD tries to reduce this set of operators, relying on a preliminary canonicalization stage. The advantage is a smaller number of cases to manage in the analysis and differentiation stages. The drawback is that the final differentiated code is also canonicalized and thus harder to read as it discards some syntactic choices of the source. As a consequence, the *Tap2Xaif* translator must apply canonicalization too, and code coming back to Tapenade may have a slightly poorer, though equivalent, form.

Tapenade's internal representation takes in Fortran (90) modules as first-class concepts, whereas they are sort of simulated in XAIF through special markers. This requires some technical manipulations in the translators on the bridges, unless the XAIF side evolves to incorporate XML elements for modules like it has for procedures.

Figure 3 presents the steps involved in using our pipeline. First, *Tap2Xaif* calls Tapenade to parse and analyze an input source code and to output the result in XAIF. Then XAIFBooster is used to generate differentiated XAIF. Then *Xaif2Tap* calls the back end of Tapenade to generate the output code from the translated XAIF.

4 The OpenAD template mechanism

Figure 1 presents a simplified view of the backend. While Tapenade fully creates the structure of the differentiated code during its differentiation stage, the XAIFBooster produces a collection of differentiated pieces (e.g. tangent, forward adjoint, backward adjoint ...) in the differentiated XAIF.

The differentiated XAIF is translated by `xaif2whirl` into a "glued" source code with special *markers* to separate out the different portions. This code is then post-processed according to so-called *templates* to produce the final differentiated code. This allows OpenAD to separate AD at the basic block level (performed by XAIFBooster) from AD at the higher flow graph level (done by the postprocessor). The basic-block level deals with differentiation of individual assignments, including for instance preaccumulation. Additionally, XAIFBooster determines what data to be checkpointed in adjoint mode, while the post process uses the template to generate checkpointing code. The postprocessor also deals refined strategies for special constructs such as fixed-point loops. In contrast, the Tapenade differentiator handles all these issues at the same time through two different Java classes, called `FlowGraphDifferentiator` and `BlockDifferentiator`, with methods from the former calling methods from the latter. This is a significant difference between the OpenAD and Tapenade.

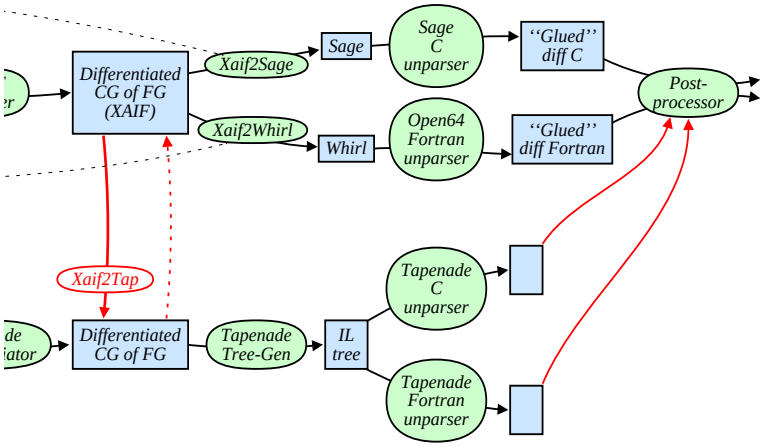

Figure 4: The back-end and the OpenAD template mechanism

Interoperation requires (see fig. 4) that Tapenade, after receiving an OpenAD differentiated XAIF file, generates the same "glued" output with the appropriate markers in it, so that it can be sent back to XAIF's post-processor at the end of the pipeline. We have tested this mechanism with success on our first examples. An alternative

could be to let Tapenade apply its own flow graph reconstruction strategy on the collection of differentiated pieces returned in XAIF. Technically, this means using Tapenade's `FlowGraphDifferentiator` while shunting the calls to its `BlockDifferentiator`. We have not implemented this alternative.

5 Conclusion and future work

We have described the implementation of a tool architecture that uses the front end, analysis, and backend of Tapenade with the transformation algorithms of OpenAD's XAIFBooster component. The new tool architecture has been tested on several small test codes from OpenAD's regression test suite. We plan to apply the pipeline to larger codes.

OpenAD and Tapenade share runtime libraries as well. Adjoinable MPI has been created to handle MPI calls in reverse mode AD. ADMM is a library to handle dynamic allocation of memory in the reverse mode of AD. It is expected the new architecture will continue to be able to use these libraries in a transparent manner.

Both OpenAD and Tapenade support non standard differentiation techniques. The Christianson method for fixed point iterations [8] is supported by both OpenAD and Tapenade. OpenAD employs a special template and its postprocessor for this purpose. Tapenade's `FlowGraphDifferentiator` handles it internally. We will will support the use of either approach in the interoperable pipeline.

Acknowledgments. This work was funded in part by support from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References

[1] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, 34(4):18:1–18:36, 2008.

[2] Jean Utke, Uwe Naumann, and Andrew Lyons. OpenAD/F: User Manual. Technical report, Argonne National Laboratory. Latest version available online at `http://www.mcs.anl.gov/OpenAD/`.

[3] L. Hascoët and V. Pascual. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software*, 39(3), 2013.

[4] L. Hascoët and V. Pascual. Tapenade 2.1 user's guide. Technical Report 0300, INRIA. Latest version available online at `http://http://www-sop.inria.fr/tropics/tapenade.html`.

[5] M. Fagan, L. Hascoët, and J. Utke. Data representation alternatives in semantically augmented numerical models. In *6th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2006, Philadelphia, PA, USA*, 2006.

[6] Paul D. Hovland, Uwe Naumann, and Boyana Norris. An XML-based platform for semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications*, pages 530–538, Anaheim, CA, 2002. ACTA Press.

[7] Jean Utke and Uwe Naumann. Software technological issues in automating the semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications, Proceedings of the Seventh IASTED International Conference*, pages 417–422. ACTA Press, 2003.

[8] Bruce Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3(4):311–326, 1994.